(54) Title: METHOD FOR TESTING AND DEBUGGING COMPUTER PROGRAMS

(57) Abstract

A simulation method allowing an experimenter to test and debug computer programs concurrently. The method utilizes the generation of signatures to observe interactions of various subprogram paths with a reference case.

METHOD FOR TESTING AND DEBUGGING COMPUTER PROGRAMS

The present invention is related to the following application filed at the same time as this application:

U. S. Patent application (PD91-0088), by Ernst Guenther Ulrich

5   and Karen Panetta Lentz, entitled **METHOD FOR MULTI-DIMENSIONAL CONCURRENT SIMULATION USING A DIGITAL COMPUTER**.

FIELD OF THE INVENTION

This present invention is related to simulating experiments on a digital computer, and in particular, to an improved method of

10   running side-by-side simulation of related experiments within one computer run.

BACKGROUND OF THE INVENTION

Based on the power of the computer and on the ability to build adequate models of reality, the simulation of experiments has

15   become an increasingly effective and often superior substitute for physical experimentation. For example, the building and testing of engineering prototypes is an experimentation effort that is done more and more in terms of models and simulations rather than conventionally.

1

Concurrent Simulation (CS) is the simultaneous, side-by-side simulation of related experiments within one computer run. CS is a method that runs on conventional computers, performing concurrent experiments without concurrent hardware. It applies and is limited to systems simulated with discrete events. Typically 10 to 1,000 times faster than serial (one-at-a-time) simulation of single experiments, its speed is largely based on the number and similarities between experiments. CS dates from 1970/1973 and was first developed for fault simulation of gate-level digital networks. Over the years it has increased in generality and, more recently, evolved into a simulation methodology. Whenever discrete event simulation is the method chosen to solve a particular problem, CS is usually better than serial simulation. CS has several advantages over serial simulation.

First, all experiments advance synchronously through the dimension of time, and CS is therefore analogous to a race in which the experiments are competitors. This constitutes a race methodology and a comparative style of simulation. This methodology and the speed of CS permit the solution of problems more dificult and larger than with serial simulation. A simulation strategy based on this methodology and comparative style is to simulate and observe related experiments which are initially the same but later become different.

2

Second, observation, which is awkward and costly for serial simulation, is handled easily and elegantly with CS. The experiments are observed comparatively, and can be compared in exact detail as well statistically. Statistical "signatures" are

5   maintained and periodically analyzed for all experiments.

Next, CS offers speed in various forms. Relative to serial simulation, experiments are compressed into a single run. The idle time between serial simulations is avoided and a simulation project is strategically accelerated. Also, due to the number of concurrent

10   experiments, due to their similarity, and the similarity between them and a reference experiment, the CPU time, as mentioned previously, is typically 10 to 1,000 times less than the equivalent serial simulations. And, based on the analysis of signatures, the initial reference experiment may often be replaced with a more

15   central one. This reduces the average differences between reference and concurrent experiments and gains additional speed.

Lastly, CS provides accuracy and generality. In fields such as biology and chemistry it is desirable to perform related and similar physical experiments in parallel, but it is normally too

20   costly due to labor, space, equipment, and the raw materials that are needed. CS is a parallel (and precisely time-synchronous) form of experimentation, and is therefore an alternative to parallel

3

physical experimentation. It requires no resources except a conventional computer and modeling/simulation skills.

4

Serial and Concurrent Simulation

The following figure shows (a) serial simulation and (b) the equivalent Concurrent Simulation.

```
     S0     S1     S2     Sn     TIME     C0=R
 5   |   .->|   .->|   .->|       1       |
     |   :  |   :  |   :  |       2       |                    C2
     |   :  |   :  d      |       3       |- - - - - ->d
     |   :  |   :  d   :  |       4       |    C1         d
     |   :  d   :  d      |       5       |- >d           d
10   |   :  d   :  d   :  |       6       |    d          d
     |   :  d   :  d      |       7       |    d          d
     |   :  d   :  d   :  |       8       |< -d           d
     |   :  |   :  d      |       9       |< - - - -  d
     |   :  |   :  |   :  |      10       |                    Cn
15   |   :  |   :  |      d      11       |- - - - - - - > d
     |   :  |   :  |   :  d      12       |< - - - - - - d
     |   :  |   :  |      |      13       |
     |__/   |__/   |__/   |      14       |
```

        (a) Serial Simulation   (b) Concurrent Simulation

5

Experiments C0 to Cn are equivalent to the serial simulations S0 to Sn. C0=R is the fully simulated reference experiment, while C1 to Cn are small scale concurrent C-experiments. For example, C2 diverges from R at time t3, exists as long as it differs from (d) from R, and converges with R at time t9. Most of the simulation work is done by the R-experiment. It handles, at no cost per C-experiment, all segments of all C-experiments identical to their to their counterparts in R. The R-experiment carries information, i.e., the identity numbers of C-experiments not performing the R-experiment. The simulation cost per C-experiment is proportional to its difference from R. If a C-experiment remains identical to R, then it is simulated at no cost. If it is almost identical to R, then it is simulated at almost no cost. Many data distinct C-experiments are handled at almost no cost by the R-experiment or as rather inexpensive fraternal C-experiments.

Testing and Debugging Computer Programs

Adequate testing and debugging of computer software is imperative for ensuring quality, yet with existing methods, the testing process is tedious and does not exercise the computer software thoroughly. This lack of thoroughness, the amount of manual labor, and the total time spent for testing and debugging a typical piece of software are shortcomings addressed here.

6

SUMMARY OF THE INVENTION

It is the object of the present invention to provide a simulation technique for the testing and debugging of computer programs. This method differs from conventional methods in that a target program is not executed but that the program's execution is simulated, and that many program path experiments are simulated simultaneously. The method here is labeled **CSPP**, the Concurrent Simulation of Program Paths. The technique and vehicle for **CSPP** is **MDCS**, Multi-Domain Concurrent Simulation. Critical input variables for a computer program are defined, and a set of input values for each of these variables is assigned. A reference value is selected and assigned a signature. Further signatures are created when input variables interact with the reference value and finally compared to correctly running cases.

Other objects, features and advantages of the invention will become apparent on a reading of the specification when taken in conjunction with the drawings in which like reference numerals refer to like elements in the several views. The objects and advantages of the invention may be realized and obtained by means of instrumentalities and combinations particularly pointed out in the appended claims. The improvements of the present invention over the prior art and the advantages resulting therefrom will become

7

more apparent upon reading the following description of the preferred embodiment taken in conjunction with the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The improvements of the present invention over the prior art

5   and the advantages resulting therefrom will become more apparent upon reading the following description of the preferred embodiment in which:

Fig. 1 is representation of a small computer program;

Fig. 2 is a table listing the terminolgy abreviations used

10   thoughout this description of CSPP;

Fig. 3 is a conceptual representation of the main parts of a computer;

Fig. 4 is a conceptual representation simulation utilizing CSPP;

15   Fig. 5 is a conceptual representation of three computer subprograms arranged to execute sequentially; and

Fig. 6 is a conceptual representation of three computer subprograms arranged to execute concurrently.

8

DESCRIPTION OF THE PREFERRED EMBODIMENTS

CSPP can simplify and reduce design verification and diagnostic fault simulation. For design verification, its CPU-time advantage over convential methods is estimated to exceed 40:1. For large systems the method of Clock Suppression may boost this beyond 200:1. For diagnostic fault simulation, the CPU-time advantage ranges from 100 to 10,000:1 over conventional methods.

CSPP is a method that utilizes MDCS to:

- Find and eliminate more bugs and find and eliminate them faster than conventionally, so that the programs tested /debugged will contain fewer residual bugs.

- Allow many test cases to be simulated in a single simulation.

- Automate portions of the testing procedure such that only viable input conditions are simulated. This eliminates the manual labor of chosing viable input conditions.

- Provide observation and automated statistics gathering of the execution of the program, such as data differences, instruction differences, and erratic behavior (infinite

9

loops). Observation includes precise coverage (exercised versus unexercised) information, as well as a precise instruction count per program path.

*   Provide the ability to simulate alternative programs against
5   each other, comparing their speeds, accuracies, and complexities.

CSSP simulates the execution of a computer program in terms of instructions or high level constructs. Its central features are that many program path "experiments" are simulated simultaneously,
10   and that unique signatures are generated for unique program paths. Each experiment is due to input conditions specified by the user. One chronic problem with conventioanl tools, the specification of nonviable input conditions and thus running of nonviable programs, is essentially avoided here because the worst types of nonviable
15   programs are automatically not executed with CSPP. Overall, CSPP is easier to use, more thorough, informative, and efficient than convential testing and debugging tools.

For program testing, the user needs to verify the correctness of (output) results of the experiments, while for program debugging
20   he would analyze signatures. Each signature, which includes a statistical "distance" between an experiment and an artificial

10

"average" experiment, is information that cannot be created with conventional testing/debugging.

CSPP is based on Multi-Domain Concurrent Simulation (MDCS), which is a generalization of Concurrent Simulation (CS). MDCS 5 permits different experiments to interact. That is, primary or P-experiments due to input variables may interact with other P-experiments to produce interaction or I-experiments. Also, MDCS automatically generates efficiency via "standin" experiments. P-experiments act in one-for-many fashion at experiment sources, 10 where many locally identical P-experiments act in a one-for-many fashion at experiment sources.

CSPP can be explained in terms of a similar but expensive alternative method. This method consists of defining input variables of a program, defining a small set of values per 15 variable, and executing the program for all orthogonal combinations of these values. For example, a program may have six orthogonal input variables and three values per variable may be defined. This constitutes a program test of 729 program versions, i.e. 3^6 input combinations. Essentially, this method is impractical because a 20 large number of non-viable program versions would be executed. CSPP achieves the intent of this method, but avoids its disadvantages.

11

For the above program a **CSPP** simulation involving six orthogonal domains is appropriate. Three values per variable are defined by the user, including one reference value. The simulation begins with the execution of the reference program or R-program.

5   This R-program then "encounters" input variables, and as a result P-programs arise which are different from the R-program. These, in turn, encouter additional input variables, and interaction or I-experiments due to these encounters are created. For example, if a program contains a bug near its entry point, its execution may

10 encounter no input variables and only one incorrect program version or path may be established. A more likely example is that input variables are encountered and that per encounter appoximately half of the specified values will generate distinct I-experiments. Thus, while a program has many potentially distinct program paths and

15 output results, it is likely that only a fraction of them will occur. The definition of different values is not restricted to input variables, but for debugging it is often useful to "force" a few different (from the R-experiment) values for internal variables.

20 If structured techniques are used to test and debug computers programs, the folowing process steps would occur:

•   designing the program

12

- calculating the complexity of all modules that make up the program
- deriving the test basis paths for a module
- deriving the data for the test paths

5   Measuring complexity quantifies the testability attributes of modules and also quantifies the number of independent test paths through a module. Knowing the complexity of a module, the user can reduce the number of paths that need to be tested. Fig. 1a illustrates an example of deriving test paths for a program. The

10   boxes containing numbers in Fig. 1a refer to the corresponding individual lines of pseudo-code in Fig. 1b. More specificly, box 1 in Fig. 1a corresponds to line 1 in Fig. 1b, box 2 in Fig. 1a corresponds to line 2 in Fig. 1b, box 3 in Fig. 1a corresponds to line 3 in Fig. 1b, box 4 in Fig. 1a corresponds to line 4 in FIg.

15   1b, box 5 in Fig. 1a corresponds to line 5 in Fig. 1b, box 6 in Fig. 1a corresponds to line 6 in Fig. 1b, and box 7 in Fig. 1a corresponds to line 7 in Fig. 1b. If the test paths were derived for the code in Fig. 1b, by enumerating all possible paths from the constructs, then they'd be the following:

20   •   1-2-3-5-6-7

      •   1-2-3-5-7

      •   1-2-4-5-6-7

13

•   1-2-4-5-7

Note, that because of the actual instructions in the code, the only viable paths are:

•   1-2-3-5-6-7

5   •   1-2-4-5-7

The intention of this example is to demonstrate that testing programs requires a lot of manual work. The test paths derived must be accurate, but notice that the user is subjected to manually distinguishing viable paths while attempting to keep the number of

10   test cases at a minimum due to central processing unit (CPU) time on a computer. As the size and the complexity of a program increases, the time of deriving test paths increases. This results in a greater number of test cases to be set up and executed.

The work described here uses a substantial amount of

15   terminology that is summarized in the table in Fig. 2.

Fig. 3 is a conceptual representation of the main parts of a computer containing and executing a program. The PC 14 is the program counter. The MAR 16 is the memory address registers. Fig. 3 further shows the connections between a Memory 10 (holding

14

programs and data) and a Network 12 that executes the program instructions. The Network 12 is the central processing unit (CPU) performing the work. In reality, the PC 14 and MAR 16 can contain only one value, pointing to one location in the Memory 10.

5   A program counter (PC) 14 and a memory address register (MAR) 16 are important nodes involved in the program flow, and 01 18 and 02 20 are therefore important observation points to be used during a typical simulation. Simularly, the data connections between memory 10 and network 12 (observation points 03 22 and 04 24) are
10   important. The basic program being observed is the reference program (R-program). Executed by the reference experiment (R-experiment), it executes reference instructions (R-instructions). Fig. 3 also shows Cs (concurrent experiments), normally injected into a memory 10 as experiment origins or into a network 12 as
15   fault origins. Initially, these latent Cs (labeled C1-lt, C2-lt, etc. in memory 10) are small differences relative to the reference experiment.

Fig. 4 is similar to Fig. 3 except that it indicates what is contained in the simulated PC 34 and MAR 36. During (concurrent)
20   simulation the PC 34 and MAR 36 may contain many values, pointing to different locations (for different experiments) in a Memory 30. More specifically, Fig. 4 shows that as the R-program exercises the latent Cs they emit C-effects and grow into "captive","fair", and

15

"strong" Cs, labeled C5-ca, C3-fa, and C4-st respectively, in network 32. These attributes describe the observability of Cs, and are useful as part of a C-signature. A C-experiment becomes increasingly observable as it produces C-effects. It could be captively observable in the network 32 (at observation points in the network), fairly observable as C-effects cross data paths O3 42 or O4 44, or strongly observable as C-effects cross the O1 48 or O2 50 control paths. In Fig. 4 the PC 34 and MAR 36 contain C-effects due to strong Cs C2-st and C4-st, and C-programs C2 and C4 are running concurrently with the R-program. Strong Cs always execute C-instructions and C-programs.

The R-program is executed by the R-experiment and all latent, captive, and fair Cs. These Cs are generally unobservable at points O1 48 through O4 46, but fair (data-only) Cs may become briefly observable. For example, as the R-program moves across O4 46 in Fig. 4, unobservable (implicit) data will move along for latent and captive Cs, but an observable (explicit) data item for fair C C3-fa 42 may also move along. This C-effect carries the ID number C3, and, without affecting the PC 34 or MAR 36, experiment C# becomes briefly observable at O4 46. This also means C3 becomes observable within the memory 30, i.e. a C-effect will be stored in a memory location.

16

A basic design verification strategy assumed here is the execution and observation of a concatenated program consisting of small subprograms. Referring to Fig. 5, a concatenated program P 60 running perfectly from beginning to end provides a strong

5   probability of the absence of design erros, a probability that increases with the number of subprograms. For each subprogram a number of related cases are executed side-by-side. In Fig. 5 subprograms P1 **62**, P2 **64**, and P3 **66** contain 8, 100, and 13 cases arranged to be executed sequentially. The R executes cases C1-0 **68**,

10   C2-0 **70**, and C3-0 **72**. All other cases are handled with Cs. Each subprogram is analogous to a race, and each case is a racer. A race is fair or strong. In a fair race only one program, i.e. the R-program is executed. It may consist of many related cases, but they differ from each other only in terms of data rather than control.

15   During a strong race C-programs are running against the R-program. C-programs arise at the beginning of the race, or later due to a strong one. A race is normally over when the R reaches the end of the subprogram. At that time the differences or similarities between cases have usually been observed. Generally, the C-

20   experiments involved in this race will then be removed. Then the R will will execute a next instruction, initiating a next race.

Observation of races can be optimized by arranging tie races, i.e., causing the final results of a race to be identical for

17

correctly running cases. This can be done for fair and strong races, and in a direct or indirect fashion. For example, arranging the tie race 99+1 = 98+2 = 97+3 = 100 is quite direct. However, if the additions 1+1 = 2 and 100+100 must be verified, this requires

5   some indirectness to create a tie. It can be done with the help of extra additions, i.e. 1+1+200 = 100+100+2 = 202. Totally unrelated cases can be forced into a tie. For example, if the predicted results of two cases are the number 7777 and the symbol ABC, a tie may be arranged with a few instructions per case: the actual

10   individual results are compared against a prestored counterpart; if they agree, a common pseudo result is stored in the same place for both cases, thus producing a tie. Design verification experiments that can be done with the above mechanism are the following:

1.   Arithmetic operations with different sets of data, e.g., A+B

15   = C, E+F = G, etc.

2.   Information transfers such as from a single memory word M1 to many registers, and subsequently to memory word M2. This is another tie race, where all correct results will be naturally identical.

20   3.   Execute related instructions opposite to each other. For example, if cases C3-0 72 to C3-2 74 in Fig. 5 contain an ADD

18

instruction as its major item to be verified, this could be
replaced by a SUBTRACT for cases C3-3 to C3-12 **76**.

4.   Execute related or unrelated instructions side-by-side. This
      is direct application of the race philosophy and is a strong
5   race. It permits side-by-side comparison of correctness and
      timing for an arbitrary number of instructions.

5.   Arrange two or more cases so that the two or more branches of
      a decision instruction will be executed concurrently. With
      proper observation, this exposes the point of departure and
10   precise timing.

      In Fig. **5** the subprograms P1 **62**, P2 **64**, and P3 **66** are
simulated sequentially. Often it will be possible to rearrange this
and simulate these programs concurrently, producing a much "shorter
but wider" simulation as seen in Fig. 5. This is efficient because
15   it reduces the clock cycles simulated. Clocks often represent the
largest share (often 90% for a large network) of simulation
activity, and thus consume a proportional share of simulation CPU-
time. Reducing the clock cycles from C to C/20 will not reduce the
CPU-time to 1/20, but may come close to it. This re-arrangement
20   demands pseudo ties, arranging it so that all correct cases have
the same result. Cases are dropped from the simulation when this

result is achieved. It shold be noted that observation is affected

here. Subprogram P2 64 in Fig. 5 may be a fair race, where only the

R-program is executed. The same subprogram P2 80 in Fig. 6 becomes

a strong race, with all caes executing strong programs distinct

5   form the reference program R = C1-0 68 in Fig. 5. This method is

also useful when a simulation must be repeated with only minor

variations to analyze a specific design problem; it facilitates the

suppression of all but the critical subprogram and thus will often

save CPU-time. Fig. 6 represents the subprograms of Fig. 5 when

10   arranged   concurrently,   i.e.   all   121   experiments   will   run

concurrently.

It should be apreciated that modifications and additions will

be apparent to those of ordinary skill in the art in applying the

teachings   of   the   invention   described   herein   to   various

15   applications. Accordingly, the invention should not be limited by

the description herein of a preferred embodiment but, rather, the

invention   should   be   construed   in   accordance with   the following

claims.

20

WHAT IS CLAIMED IS:

1   1.   A method for testing and debugging computer programs utilizing

2   multi-domain concurrent simulation comprising the steps of:

3   a.   automatically establishing a set of program paths within

4   each computer program;

5   b.   generating a signature from the simulation of each

6   program path;

7   c.   comparing the signatures generated from each program path

8   to the expected execution results to determine the

9   absence, presence, and location of any number of program

10   design errors; and

11   d.   comparing two or more computer programs for the purpose

12   of optimization.

1   2.   A method for testing and debugging computer programs utilizing

2   multi-domain concurrent simulation on a digital computer

3   system comprising the steps of:

21

4   a.   defining critical input variables for a computer program;

5   b.   defining a set of values for each of the critical input
6   variables;

7   c.   identifying a reference value from the first value
8   assigned to each input variable;

9   d.   initiating a simulation run on a computer with the
10   reference value and assigning it a unique signature;

11   e.   creating a group of primary experiments from the
12   interaction of input variables with the reference value
13   and assigning them each with unique signatures;

14   f.   creating a group of interaction experiments from further
15   interaction between input variables and primary
16   experiments thus creating further unique signatures;

17   g.   analyzing the signatures generated from each program path
18   to the expected execution path results to determine the
19   absence, presence, and location of any number of computer
20   program design errors;

21   h.   performing obseravtions via signature analysis; and

22   i.   arranging and performing optimization by causing the
23   final results of a simulation run to be identical for
24   correctly running cases.

23

1/5

J = 1;
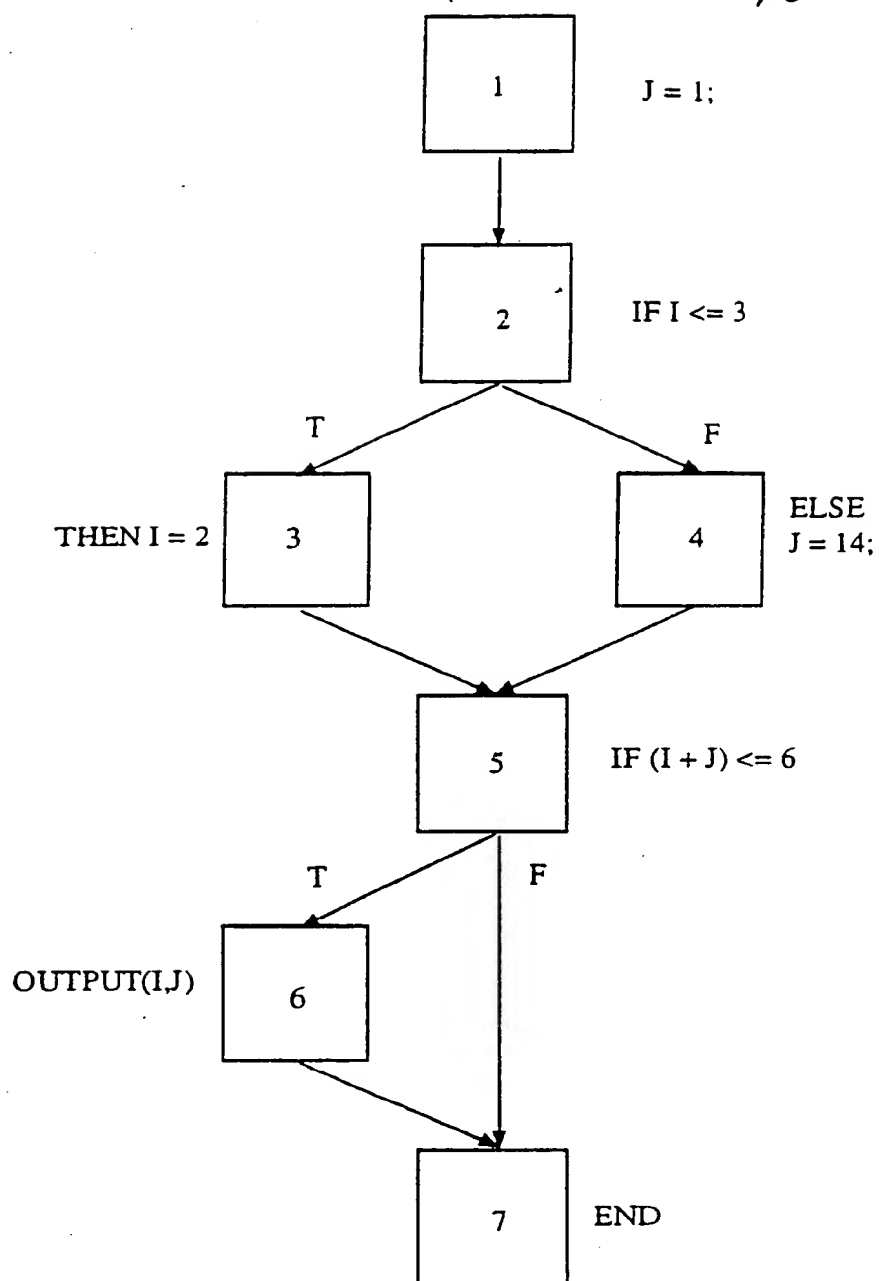
IF I <= 3

```
1   J = 1;
2   IF I <= 3
3      THEN I = 2
4        ELSE J = 14
5   IF (I + J) <= 6
6      THEN OUTPUT (I.J);
7   END
```

FIG. 1b

T   F

THEN I = 2   ELSE
                                   J = 14;

IF (I + J) <= 6

T   F

OUTPUT(I,J)

END

Complexity: V(G) = 3

FIG. 1a

C = CE, a Concurrent or C-experiment

CS = Concurrent Simulation

CSPP = Comparative Simulation of Program Paths

C-effect = An effect due to a C-experiment (also C-item)

C-origin = An origin or cause of a CE

C-program = A program executed by a CE

Captive C = A C not "escaping" from the network

C-R distance = A C-to-R similarity measure

C-C distance = A C-to-C similarity measure

Distance = Statistical difference between two experiments

Fair C = A C not executing a C-program

Fair Race = A race involving only fair Cs

Explicit Simulation = Work done for an individual C

Implicit Simulation = Work done by the R for all Cs

Latent C = A C not producing any C-effects

MAR = Memory Address Register

Meta-program = An artifical "observation" program

PC = Program Counter

Primary Observers = Normal user defined observation points

Race = The R and Cs running side-by-side

R = RE, the Reference Experiment

R-program = The program executed by the R

Secondary Observers = Auxiliary observation points

Signature = Statistical information about an experiment

Strong CE = A CE that executes a CE-program

Strong Race = RE and CE programs running side-by-side

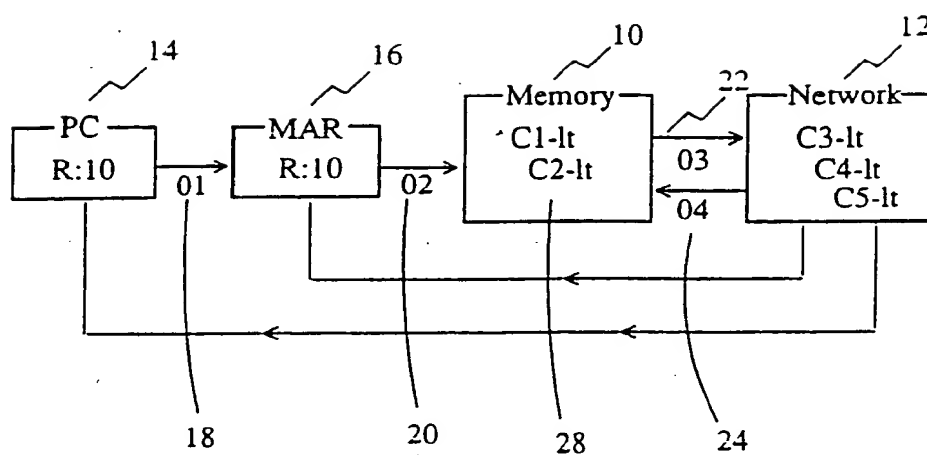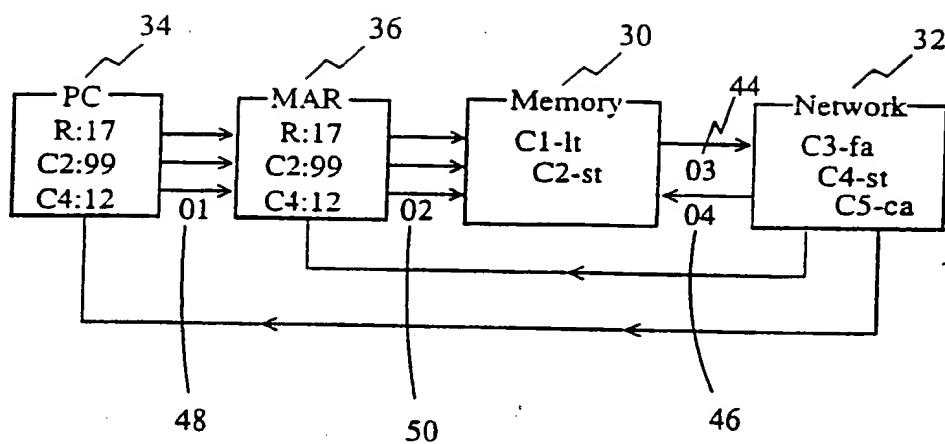Tie = The RE and CEs produce the same result

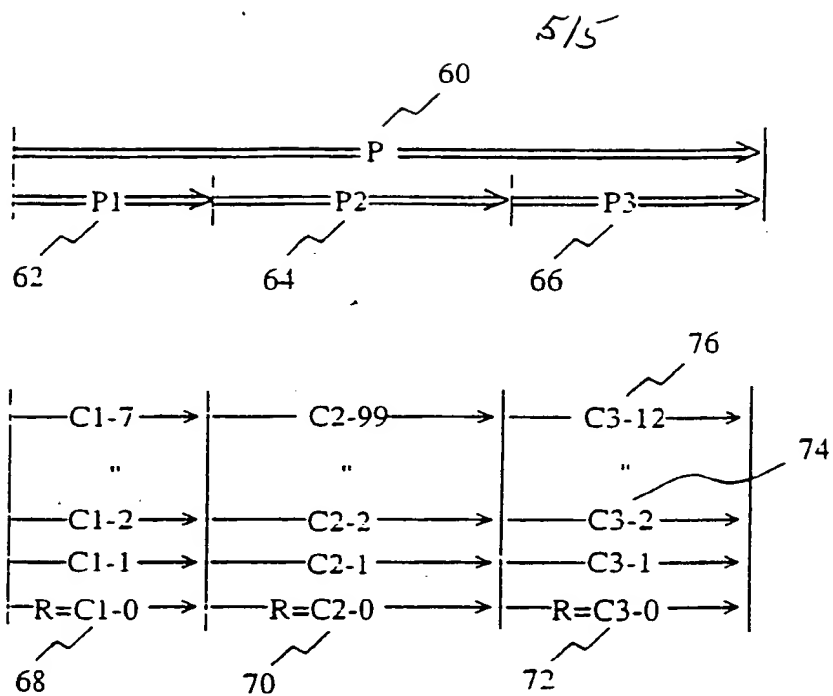# FIG. 2

3/5

FIG. 3

4/5

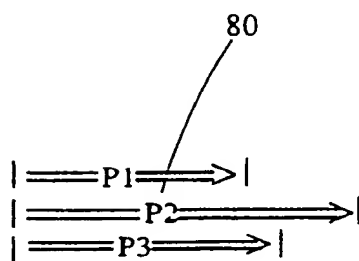FIG. 4

5/5

FIG. 5

FIG. 6

I. CLASSIFICATION OF SUBJECT MATTER   (if several classification symbols apply, indicate all)[6]

According to International Patent Classification (IPC) or to both National Classification and IPC

Int.Cl. 5 G06F11/00

II. FIELDS SEARCHED

Minimum Documentation Searched[7]

Classification System	Classification Symbols
Int.Cl. 5	G06F

Documentation Searched other than Minimum Documentation
to the Extent that such Documents are Included in the Fields Searched[8]

III. DOCUMENTS CONSIDERED TO BE RELEVANT[9]

Category°	Citation of Document,[11] with indication, where appropriate, of the relevant passages[12]	Relevant to Claim No.[13]
Y	EP,A,0 179 334 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 30 April 1986 see column 2, line 48 – line 63	1
A		2

Y	GB,A,1 413 938 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 12 November 1975 see page 2, line 8 – line 45	1

° Special categories of cited documents :[10]

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

IV. CERTIFICATION

Date of the Actual Completion of the International Search	Date of Mailing of this International Search Report
06 JULY 1992	1 3 JUIL. 1992

International Searching Authority	Signature of Authorized Officer
EUROPEAN PATENT OFFICE	Guido Corremans

Form PCT/ISA/210 (second sheet) (January 1985)

This annex lists the patent family members relating to the patent documents cited in the above-mentioned international search report.
The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information. 06/07/92

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP-A-0179334	30-04-86	US-A- 4729096 CA-A- 1223663 JP-A- 61103247	01-03-88 30-06-87 21-05-86
GB-A-1413938	12-11-75	None	

EPO FORM P0479